Vol.2009-ARC-184 No.17
2009/8/5

マルチコアおよび GPGPU 環境における画像処理最適化

矢野勝久† 境隆二†
高山征大† 出宮健彦†

スケーラを題材として，マルチコアおよび GPGPU 各々の HW 特性に適した画像処理の最適化を図る．マルチコア環境では，数値演算処理の削減，SIMD 化など直列性能の最適化を行った後, OpenMP を利用して並列化を図る. GPGPU（CUDA）では，スレッド並列を優先して並列処理の設計を行いブロックサイズを決める．また，CUDA 特有のメモリ階層に適したメモリアクセスの最適化を図る．

Imaging optimization for multi-core and GPGPU System

Katsuhisa YANO† Ryuji SAKAI†
Motohiro TAKAYAMA† Takehiko DEMIYA†

We evaluated image scaling algorithm optimizations for multi-core processors and GPUs exploiting their HW specialties. On multi-core system, we first did sequential optimization including reduction of numerical processing and SIMD vectorization, then applied OpenMP for parallelization. On GPGPU (CUDA), we took thread-level parallelization approach to decide the appropriate block size and designed memory access pattern suitable for the CUDA specific memory hierarchy.

1. はじめに

近年，x86 等の CPU では動作周波数の頭打ちなどにより直列性能の大幅な向上は望めなくなってきており，処理能力の向上はマルチコア化による並列性能改善の方向にシフトしてきている．

一方，GPU の方は，多くのプロセッサを搭載することによって，ここ数年処理能力を飛躍的に向上させてきたが, その適用分野はグラフィック処理関係に限られていた．しかし，NVIDIA CUDA（Compute Unified Device Architecture）[1]の登場により，C/C++ 言語を使って GPU 上で容易に汎用演算を記述できるようになってきた．

既存のプログラムを CUDA 上で動作させることも比較的容易であるが, CUDA 上で十分な性能を引き出すには，CPU 上の最適化とは異なる GPU 特有のノウハウが必要となってくる．

本報告では, 画像処理を題材にして, CPU および GPU 各々に適した最適化を施し，評価を行っていく．

2. 評価環境

本報告の評価は下記の環境で行った．

CPU	Intel Core i7 2.93GHz
GPU	NVIDIA GTX 280
GPGPU	NVIDIA CUDA SDK V2.2

表 1 評価環境

画像処理の例としては，画像の拡大処理を行うスケーラを取り上げる．また，スケーラのアルゴリズムとしては，一般的によく知られ東芝超解像[2]の仮拡大処理でも利用されている３次畳込み内挿法（Cubic Convolution[3]）を選択した．

† （株）東芝ディジタルメディアネットワーク社コアテクノロジーセンター
Core Technology Center of TOSHIBA CORPORATION Digital Media Network Company

3. マルチコア環境における画像処理最適化

3.1 直列処理の最適化

３次畳込み内挿法（Cubic Convolution）は，変換後の画素の座標位置に対応する変換元の４x４点の画素に重み付け演算を施して算出する．

まず評価のベースとなるプログラムとして，３次畳込み内挿法を下記のような一般的な最適化手法を使って実装した．

1. 重み付け係数のテーブル化
2. 演算結果の再利用による演算処理回数の削減
3. SIMD 化

このプログラムの SD（720x480）→HD（1920x1080）変換における処理時間は以下の通りである．

	フレーム処理時間
重み付け係数テーブル化	37.2 ms/frame
演算結果の再利用	13.4 ms/frame
SIMD 化	5.8 ms/frame

表 2　スケーラ直列性能（SD→HD）

x86 CPU では浮動小数点演算性能は高くないが，SIMD を利用しない C 言語のままでも 75fps 程度のフレームレートが出ている．また，SIMD 化を行った場合，直列処理だけでも 170fp 程度の性能が得られた．

3.2 マルチコアによる並列処理

上記の SIMD 化を行ったプログラムをベースに，マルチコアに対応した並列化を図った．並列処理 I/F は，話を一般化するため今回は OpenMP を利用した．Intel Core i7 は物理コア数 4 の Quad Core CPU である．Core i7 は論理コアであるハイパースレッドも利用可能であるが，スレッドの挙動が不安的になるため今回の評価ではハイパースレッドは OFF にした．同様の理由で，ターボモード（動作していない CPU コアが存在する時に，動作クロックを一時的にあげる）も今回の評価では OFF にしている．

スレッド数	処理時間	比率
１スレッド	5.8 ms/frame	－
２スレッド	2.9 ms/frame	2.0 倍
３スレッド	2.0 ms/frame	2.9 倍
４スレッド	1.5 ms/frame	3.9 倍

表 3　スケーラ並列性能（OpenMP）

1 スレッドでの処理時間を基準にすると，2～4 スレッドではスレッド数にほぼ比例して処理時間が短縮している．この３次畳込み内挿法の例では，変換先の画素値算出で処理の依存関係がないため，OpenMP でも良好な並列処理性能が得られる結果となった．

3.3 HD→4K2K スケーリング

上記で最適化したプログラムを HD（1920x1080）→4K2K（3840x2160）スケーリングに適用すると以下のような結果になった．

	処理時間（1 スレッド）	処理時間（4 スレッド）
重み係数テーブル化	216 ms/frame	54.8 ms/frame
演算結果の再利用	54.5 ms/frame	14.3 ms/frame
SIMD 化	23.5 ms/frame	5.9 ms/frame

表 4　HD→4K2K スケーリング性能

HD→4K2K スケーリングの処理時間は直列処理，並列処理共に SD→HD スケーリング処理の約 4 倍の処理時間となっている．4 スレッド（SIMD）では 170fps 程度の処理能力があることが確認され，実時間処理が十分可能なことがわかった．

4. GPGPU（CUDA）環境における画像処理最適化

CUDA を実行可能な NVIDIA の GPU では，８つのスレッドプロセッサを１組としたマルチプロセッサによって並列処理が実現される．今回，評価で使用した GTX 280 では 240 個のスレッドプロセッサが搭載されている．

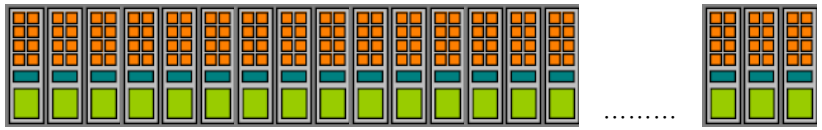

図 1　NVIDIA GPU スレッドプロセッサ

4.1 ブロック並列

まず，マルチコア環境の評価で使用したＣ言語プログラムをベースにして，CUDA のブロック並列を利用しない直列プログラムを作成し，性能を測定した．

	処理時間
C 言語直列（CPU）	37.2 ms/frame
ブロック並列なし	3250 ms/frame

表 5　CUDA 初期性能

CUDA でブロック並列をしない，つまりスレッドプロセッサによる並列処理を全く行わない状態で性能を測定すると，最適化を施していない CPU の直列処理よりさらに 80 倍以上遅くなり，実用性が全くないことがわかった．

次に画素単位にブロック並列処理（1920x1080 ブロック）を行うプログラムを作成して，性能を測定した．

	処理時間
ブロック並列なし	3250 ms/frame
画素単位でブロック並列	25.2 ms/frame

表 6　CUDA ブロック並列性能

ブロック並列を全く行わない場合と比較して 120 倍以上高速になり，ブロック並列だけでも GPU 上の多数のスレッドプロセッサを活用できることがわかった．ただし，性能の絶対値としては，ようやく C 言語（CPU）の直列処理を上回るようになった程度である．

4.2 スレッド並列

次に各ブロックに複数のスレッドを割り当て，ブロック毎にスレッド数分の画素を処理するようにした．

スレッド数／ブロック	処理時間
4	6.8 ms/frame
16	2.1 ms/frame
64	1.0 ms/frame

表 7　CUDA スレッド並列性能

今回の Cubic Convolution の例では，16 スレッドまでは概ねリニアに性能がスケールし，64 スレッドまでなだらかに性能が改善した．64 スレッド以上では，処理時間はほとんど変わらなかった．64 スレッド／ブロック時の処理時間は 1ms（1000fps）程度で，ブロック並列を全く行わなかった時と比較して 3000 倍以上高速になっている．

以上のことから，CUDA で高い並列性能を得るには，ブロック並列，スレッド並列は必須で，スレッド並列の設計を優先させてブロックサイズを決めた方が良い結果が得られることがわかった．

4.3 メモリアクセス最適化

CUDA は下記のような異なる特性をもったメモリ階層で構成されている[4].

1. グローバルメモリ
2. 共有メモリ
3. ローカルメモリ
4. テクスチャメモリ，コンスタントメモリ

このうち，高速にメモリアクセスをすることのできる共有メモリの使いこなしが最適化のポイントなるが，共有メモリにはサイズ制限がある．また，共有できるのは同一ブロック内上のスレッド間だけであり，異なるブロック間で共有メモリを参照することはできない．さらに，異なるカーネル間で共有メモリを保持し続けることもできないため，カーネルの実行単位で必ず一度はグローバルメモリなどから共有メモリに

データをロードする必要がある．このため，アクセス頻度の低いデータの場合，共有メモリを利用しない方が高速になるケースもある．

また，メモリアクセス制御関連で最適化のポイントとなるのは，グローバルメモリにおけるコアレスアクセス[4]と，共有メモリにおけるバンク競合回避[4]である．一般に CPU 環境では，データがキャッシュに載りやすくするため，データ構造や実行処理単位の局所性をあげることが多いが，CUDA ではコアレスアクセスとなるようにデータの配置を調整した方が速くなるケースもある．

前節では画素の処理単位でスレッドを割り当てたが，画像処理ではメモリアクセス関連のコストが高いため，メモリアクセスの効率化を優先させてスレッドを割り当てた方が良い結果を生むことも多い．

複雑な数値計算を行う場合，CPU の最適化では演算の一部をテーブル化してデータとして持つケースがあるが，GPU の場合テーブルからデータを読み取るコストより演算コストの方が低い場合もある．

これらのメモリアクセス関連の最適化を施したプログラムを作成し，性能を測定した結果が以下である．

	処理時間
ブロック並列なし	3250 ms/frame
ブロック並列	25.2 ms/frame
スレッド並列	1.0 ms/frame
メモリアクセス最適化	0.35 ms/frame

表 8 CUDA メモリアクセス最適化

各種最適化を施した結果は，並列処理をまったく行わない場合と比較して，実に 10,000 倍近くの性能差がある．GPU では多くの演算プロセッサを有効活用することによって高い並列性能が得られるが，半面最適化が不十分であると性能が大きく低下することになる．

5. 今後の課題

5.1 CPU,GPU 協調動作の検討

これまでの評価で，マルチコア環境，GPGPU 共に高い処理能力あることが確認できた．マルチコアおよび GPGPU を同時に並列実行することによってさらなる処理能力の向上が期待されるが，CPU と GPU を効率よく協調動作させるには，CPU と GPU 間のデータ転送コストというオーバーヘッドが課題となる．下記は SD サイズ画像，HD サイズ画像の CPU,GPU 間のデータ転送コストである．

CPU→GPU（SD）	0.23 ms/frame
CPU←GPU（SD）	0.17 ms/frame
CPU→GPU（HD）	1.0 ms/frame
CPU←GPU（HD）	0.7 ms/frame

表 9　CPU⇔GPU 画像データ転送コスト

特に HD サイズ画像の転送コストは 1ms であり，GPGPU（CUDA）で最適化されたスケーラ処理本体（0.35ms）と比較しても 3 倍近くの大きなコストとなっている．非同期のデータ転送 I/F を利用することにより，データ転送自体はバックグランド処理することも可能だが，今回のスケーラの例では，転送コストの低い SD サイズの変換元画像データを GPU に転送し，スケーリング処理を全て GPU 側でした方が有利である．特に変換元画像データが最初から GPU 側に存在するケースでは，協調動作による性能の上積みは期待できない．

6. まとめ

1. マルチコア環境では，直列処理の性能改善を十分行ってから，並列化を図ることによって，より高い処理能力を得ることができる．
2. 依存関係の少ない画像処理では，OpenMP でも物理コア数にほぼ比例した十分な並列性能を得ることができる．
3. CUDA ではブロック並列処理，スレッド並列処理は必須である．特にスレッドの並列処理を優先させて並列化の設計を行い，ブロックサイズを決めた方が良い結果が得られる．
4. CUDA で画像処理のように大規模なデータを扱う場合は，メモリアクセス関連の最適化が重要である．この際，グローバルメモリのコアレスアクセスと共有

Vol.2009-ARC-184 No.17
2009/8/5

メモリのバンク競合回避に着目して，データ構造の設計やデータの再配置を行うことで性能を向上させることができる．さらに，メモリアクセス処理を優先させたスレッドの割り当てや演算処理順序の最適化によっても性能を改善することが可能である．

参考文献

1 http://www.nvidia.co.jp/object/cuda_home_jp.html

2 井田,松本,五十川:画像の自己合同性を利用した再構成型超解像,信学技報, vol.107,no. 380, IE2007-135,pp. 135-140,2007.
松本信幸,井田孝:フレーム内再構成型超解像の領域適応処理による高画質化の検討, IE2008-6, pp. 31-36 2008,

3 R. G. Keys, “Cubic convolution interpolation for digital image processing,”IEEE Trans. Acoust. Speech, Signal Process, vol. 29, no. 6, pp. 1153–1160, 1981.

4 NVIDIA CUDA Programming Guide Version 2.2